Enhancing Programming Understanding through Conceptual Schemas in Introductory Courses

Thais Helena Chaves de Castro[1], Crediné Silva de Menezes[2], Alberto Nogueira de Castro Junior[1], Rosane Santos Caruso de Oliveira[2], Maria Cláudia Silva Boeres[2]

[1]Computer Science Department – Federal University of Amazonas (UFAM)
Av. Gal. Rodrigo O. J. Ramos 3000 – 69.077-900 – Manaus – AM – Brasil

[2]Informatics Department – Federal University of Espírito Santo (UFES)
Av. Fernando Ferrari, s/n – 29060-970 – Vitória – ES – Brasil

{credine,rcaruso,boeres}@inf.ufes.br

{thais, albertoc}@dcc.fua.br

Abstract. This paper discusses an experience with programming courses using identification and formal representation of programming schemas and their potential for automatic analysis. A relation with Bloom's taxonomy has been used to support the classification of these schemas. We intend to use these tools for classification as well as feedback routing with respect to source code produced by programming students.

Keywords: programming education; program patterns; knowledge representation.

1. Introduction

Computer programming is one of the most elaborate cognitive tasks. To the student of the matter is required a precise vision of each problem addressed, knowledge of an appropriate formalism and the ability of combining the adequate elements for building a formal solution. Construction of abstractions and use of artificial languages are examples of the many elements involved in this process. These aspects have been a big challenge to teachers of introductory programming courses [Mckeown et al, 1999; Joosten et al, 1993; Giegerich et al,1999].

It has been observed that during initial training in activities related to artifact production a valuable factor is the prospect of the student receiving criticism, comments and recommendations about the solutions presented by him (feedback). In programming, that goal is reached partially through the use of programming languages for which an environment of compilation/interpretation exists. The syntactic analyzer produces the criticism with regard to the syntactic errors and it could be able to address some inadequacies, as for example, the absence of variables' initialization. On the other hand, upon submitting a program to execution, the student can compare the actual results with the expected ones and so evaluate the correction of his program. Some environments can even perform these comparisons from a test plan. Thus, machine-supported feedback is feasible and has already been used.

However, the kind of answer the computer can supply, although helpful, represents barely a small part of the criticism and orientation needed for reaching expertise in the art of computer programming. In the first place, the interpreter does not match the results produced by a program with the expected product, that is, nothing is said about the correction of the program. Secondly, nothing is said about the quality of the obtained solution. Although for the first topic one can already count on tools for the verification of results, it is important to mention that the second topic establishes a big challenge for teachers and tutors of introductory courses. It is known that quantity and range of solutions to be analyzed depend on the ability of the students in constructing them and of

the nature of the resources used. Analysis and evaluation of student's solutions are very important for him to reach proficiency, helping him to get used to be questioned about his product, which certainly, still lacks an adequate model.

Usually, introductory courses are characterized by the absence of proper analysis for the solutions presented – a real danger for the education of a competent professional. In general, a student gets very satisfied when his program is working and running apparently in a correct way. Ideally, student's solution should be keenly criticized, clearly showing the kind of inadequate choices were made, regarding the quality of the presented program. Consequently, the task of evaluation becomes quite strenuous, given that, for the satisfactory development of student's expertise is necessary that he solves a significant quantity of programming problems.

The work reported here is part of a multi-institutional effort that seeks definition of paths for minimizing the task of evaluation. We search for the elements needed for the construction of assistants capable of analyzing presented solutions, classify and answer them with remarks suitable for helping student's comprehension, as suggested in [Castro et al, 2002]. From analysis of a collection of problems solved by students, a classification of solutions and their variations was done, allowing us to develop a conceptual structure. We have formalized this knowledge in order to have a description of the knowledge to be used by software agents. In this work we discuss upon some programs developed by students in their first programming course. These programs were written in Haskell, a functional programming language [Bird & Wadler, 1988].

2. Identifying Program Schemas

In order to identify the classes of solutions presented by students, the initial phase of this work consisted of analysing students' answers to several sets of exercises from introductory disciplines of functional programming, from both Computer Engineering and Computer Science courses in two of Brazilian federal universities during three academic semesters. Six groups of students from one institution and two from the other participated in the research.

From the beginning, we observed that there was no need to verify all exercises (that would require quite a big effort with no significant gain), we found to be possible to identify program patterns using only a subset of the exercises. Considering the goals of our work, only those solutions that produced correct results for a test plan were considered. For each exercise selected among the original sets, the responsible teacher built a solution, named *base solution*, considering the maturity level of the students and the concepts that had been presented and discussed in the classes. In addition to each base solution, teachers also estimated deviations of it, resulting in an initial set of solutions for each exercise.

A careful analysis in the whole set of base solutions and their deviations was carried out in order to eliminate redundancy and producing a list of distinct solutions. Using the base solution as reference, each solution submitted by the students was analyzed, in the search of elements that might have sat differences between them, allowing a tree of the variations found for each exercise, been built. In some cases, the solution of a student lead to the identification of a new solution for a problem (or a style variation), and when this occurred, the initial solutions were redefined and the solutions for that item, reclassified.

2.1. The Haskell's subset considered

For the study described here, we have considered only topics covered in the initial stage of the course, consisting of the language subset represented through the following grammar.

<definition> : : = <name> {parameter} **=** <expression> [<local definition>]
<expression> : : = <expression> | <conditional expression>
<conditional expression> : : = **if** <expression> **then** <expression> **else** <expression>
<local definition> : : = **where** <definition>

3. Conceptual Schemas – the nature of the knowledge

As a result of successive phases of classification and organization on solutions provided by the students, it was possible to classify the variations of solutions into **syntactic variations** and **cognitive variations**. Syntactic variations refer to the differences between the forms of construction obtained from the possibilities allowed by the programming language. Cognitive variations are related to different mental models that students build to elaborate solutions for the same problem. It has been noticed also semantic variations, which will not be analyzed in this work because it is not the main focus of this research. Now, we will discuss a little more on these two kinds of variations and address the use of ***evidence*** as a strategy for obtaining the semantic variation.

3.1. Syntactic Variations

It is important to identify syntactic variations of a solution so we might identify equivalent solutions to the base solution. The main syntactic variations identified were:

i. The unnecessary use of the <if...then...else>, when the expression was of the boolean type. Ex.:

`f x = if x>0 then True else False,` when it would be enough to use

`f x = x>0;`

ii. The unnecessary use of parenthesis. Ex.:

`if x>0 then (if...) else ...,` where it would be enough to use

`if x>0 then if ... else ...;`

iii. The differentiated use of the <if...then...else> concerning to the domain of interpretation of the problem.

Example 1: `if x>0 then 1 else 0` or

`if x<0 then 0 else 1.`

Example 2: `if x>0 then <exp1> else <exp2>` or

`if not x>0 then <exp2> else <exp1>;`

iv. The use of the operators' commutability. Ex.:

`<exp1> && <exp2>` it is similar to `<exp2> && <exp1>`

3.2. Cognitive Variations

Cognitive variations are directly concerned with the way students organize theirsolutions. In that point of their learning, it is expected that the concepts present in their solution are directly related with the concepts found in the problem description or that can be easily derived of it.

From a more general point of view, we can identify the existence of different metaphors for a given problem. Each metaphor is determined by a set of concepts. The student can assimilate these concepts in different levels of abstraction, devising, from that, a range of solutions. We believe that perception of abstractions at an adequate level will produce solutions of better quality. Therefore, it is of great relevance for the student to be encouraged and recommended to know and use that.

In order to place a theoretical basis for the research reported in this work, we have defined a correspondence with Bloom's taxonomy [Bloom, 1956], in aspects concerning with cognitive competences (levels of abstraction) related to evidence of typical abilities at each level. These competences are:

a. knowledge

b. understanding

c. application

d. analysis

e. synthesis

f. evaluation

These levels of abstraction pinpoint typical phases of proficiency as illustrated by the following scenario: When someone studies a subject for the first time, a History topic for example, that person is able to grasp dates, events and places, and possibly a general (superficial) vision of the issue studied, he is in the most basic phase of the cognitive development (a). When that student go deeper in the readings, he begins to get the understanding of the questions related to the theme as well as its meaning, and may even be able to predict their consequences (b). As the reading progresses, the student would be apt to use the information obtained and apply them it in another context. At this stage, he has reached the phase (c). The two following phases occur almost simultaneously. In the first one (d), the student recognizes the hidden meanings (he would read "between the lines"), and can identify new components; in the second one (e) he can make generalizations from known facts and is already able to devise conclusions. In the last phase of the cognitive development (f) the student compares and discerns ideas, makes choices based on reasoning, recognizes the subjectivity of facts and is able to valuate evidences.

Based on these cognitive abilities, hence called competences, we have grouped programs with the same level of abstraction, resulting in the following classification:

i. absence of record

ii. highlight

iii. naming

iv. parameterization

v. generalization

The situation where the student does not leave any clue of having realized the concept is called absence of record (i), and is related to the competence (a). The first indication that the abstraction was taken in account is given when the student uses mechanisms of priority levels through parentheses (ii), related to the competence (b). By giving a name to his artifact, the student makes clear, mainly by the choice of this name, that an abstract concept was captured (iii), and this provides opportunity for using the same concept more than once, an action related to the competence (c). With "parameterization" (iv), the student reaches the highest level of concept perception and of the interaction between concepts, related to competences (d) and (e). By defining parametric functions, the same concept can be used in many situations in a flexible way. Generalization (v) can support program writability, but can also produce quite confusing codes with low readability. Careful evaluation, mentioned as competence (f), would then be needed.

Sometimes it is not clear in the solution presented by the student, how its main elements would relate to the domain of the problem. It is also common student solutions to show no concern with efficiency or readability. These situations can be better illustrated through the set of solutions for a specific problem, shown in Table 1.

Looking at solutions in Table 1, it is possible to see that solutions 3 and 4 emphasize the fact that *a* and *b* share the same property. This it is not shown clearly in solutions 0, 1 and 2. Solution 0 presents a simplistic view of the problem, not grasping relevant abstractions of it. That solution does not show clearly the idea of *range*, an important abstraction to that problem. As a good solution should embody information about the problem, and that does not happen in solution 0, it is considered *inflexible*. Solution 1 shows an abstraction for range, giving indications that the student perceived the relevance of that concept. In solution 2 we can see not only the abstraction, but also its naming. Solution 3 presents a parameterization, suggesting perception to reuse. Finally, solution 4 gives a comprehensive generalization for the problem presented, including the abstractions identified in the previous solutions. That solution also includes a range handling through a function describing ranges (belongs_to).

Table 1 – Cognitive variations for a problem

Problem 1 – to evaluate whether two numbers, *a* and *b*, are in the interval [0..6]	
Solution 0	Absence of record
Prob1 a b = a>=0 && a<=6 && b>=0 && b<=6	
Solution 1	Highlight
Prob1 a b = (a>=0 && a<=6) && (b>=0 && b<=6)	
Solution 2	Naming
prob a b = prob1 && prob2 where prob1 = a>=0 && a<=6 prob2 = b>=0 && b<=6	
Solution 3	Parameterization
prob a b = prob1 a && prob1 b where prob1 x = x>=0 && x<=6	
Solution 4	Generalization
belongs_to x ni nf = x>=ni && x<=nf -- prob a b = prob1 a && prob1 b where prob1 x = belongs_to x 0 6	

3.2.1. Generalization × Reuse

Solution 4 in Table 1, offers some advantages over the others. These advantages can be considered from the point of view of a reader trying to understanding that solution (readability) or from the need of devising solutions for new problems (reuse). We illustrate this context by submitting modifications to the original problem. For each new instance of the problem we present a new solution, showing up the ease in which the script is adjusted for attending the new set of requirements. By *script* we mean a set of function definitions, needed to solve a specific problem.

i. Transformation of problem 1 by defining another closed interval (Table 2).

Problem specification: to evaluate whether two numbers, *a* and *b*, both are in the closed interval [2..8].

Table 2 – Value modification

Solution 4a
belongs_to x ni nf = x>=ni && x<=nf -- prob a b = prob1 a && prob1 b where prob1 x = belongs_to x 2 8

ii. Transformation of problem 1 by defining for an an open interval (Table 3).

Problem specification: to evaluate whether two numbers, *a* and *b*, both are in the open interval (0..6).

Table 3 – Open interval

Solution 4b
belongs_to x ni nf = x>ni && x<nf -- prob a b = prob1 a && prob1 b where prob1 x = belongs_to x 0 6

iii. Transformation of problem 1 by including another parameter (Table 4).

Problem specification: to evaluate whether three numbers, *a*, *b*, and *c* are all in the interval [0..6].

Table 4 – A three parameters function

Solution 4c
belongs_to x ni nf = x>=ni && x<=nf -- prob a b c = prob1 a && prob1 b && prob1 c where prob1 x = belongs_to x 0 6

3.2.2. Inconvenient Generalizations

Although the generalization process was very interesting for the case presented on the earlier section, its use cannot always be considered appropriated. The following example illustrate on of these situations.

Problem 2: Given 3 numbers, *a*, *b* and *c*, to determine the arithmetic average of the extremities.

It's easy to notice that the problem states, in its definition, two constant values: (i) the amount of numbers to be considered (*n*) and, (ii) the amount of numbers we will determine the arithmetic average.

In this case, we can observe the following aspects:

- The value of *n* interferes with the amount of if-then-else structures to be used;
- At this stage the student does not know much about language resources, what keeps him from using more generic constructions;
- Thus, generalization is possible for the problem domain, but is not taken in the solution domain due to the absence of resources;
- In this case, it is needed to constrain on problem's data.

For the problem domain presented here, two relevant concepts could be easily inferred from a presented solution: the smallest number and the biggest number. There is, obviously, the abstraction *arithmetic average*, as shown on Table 5. It's usual, also, to get less elegant solutions that use parentheses, as shown on Table 6.

Table 5 – Abstraction for "smallest" and "biggest"

Solution 2a
averageOfExtremes a b c = aritAverage where aritAverage = (small + big) / 2.0 small = if a<b then if a<c then a else c else if b<c then b else c big = if a>b then if a>c then a else c else if b>c then b else c

Table 6 – Use of parentheses

Solution 2b
averageOfExtremes a b c = ((if a<b then if a<c then a else c else if b<c then b else c) + (if a>b then if a>c then a else c else if b>c then b else c)) / 2.0

3.3. An strategy for improving readability

Analysis over student programs have shown some cognitive variants, and one of the most interesting is the one we called "evidentiation". An evidentiation is the isolation of a term that occurs several times within a description, like in the following expression, where multiplication by *b* happens a number of times.

a * b + (c + d) * b + h + (e + f – 3) * b + g

To put the term '* b' in evidence would lead to:

h + g + [a + (c + d) + (e + f – 3)] * b

Let us see another example, this time with a program written in Haskel, shown on Table 7.

Table 7 – Half of the smallest number

Problem 3: Given two numbers, *a* and *b*, find the half of the smallest of them.	
Solution 0	Comment
prob3 a b = if a<b then a / 2.0 else b / 2.0	An attentive student will soon realize that this solution could be rewritten by putting in evidence the division by 2.0, leading to solution 1.
Solution 1	Comment
prob3 a b = (if a<b then a else b) / 2.0	This solution emphasizes the abstraction "smallest between two numbers" through the expression within parentheses. If this concept is named, we got the next solution.
Solution 2	Comment
prob3 a b = smaller / 2.0 where smaller = if a<b then a else b	Further refinements would eventually lead to solution 3.
Solution 3	Comment
smaller x y = if x<y then x else y -- prob3 a b = smaller a b / 2.0	A superior refinement was reached.

In the example above, starting with isolation of the division by 2.0, a generalization was reached by focusing on the structure for *smaller*, a concept that previously was not made explicit. Although the kind of solution in this example would eventually be obtained by identifying relevant concepts from problem domain, we believe that evidentiation might be a good way of searching for hidden concepts. Thus, in the strenuous activity of developing suitable solutions, evidentiation must be set as a helpful *heuristic*.

We also have noticed strong indications that evidentiation would be a valuable tool when working in the domain of solutions, supporting the search for those with better performance. For example, in the expression shown in Table 7, a reduction on the amount of multiplication carried out was obtained at the first steps.

We illustrate this context a little further by following again successive refinements of solutions for Problem 2, as shown in Table 8.

Table 8 – Another go at Problem 2

Problem 2: Given 3 numbers, *a*, *b* and *c*, to determine the arithmetic average of the extremities.
Solução 2a
averageExt a b c = if (a>=b) && (a>=c) then case1 else if (b>=a)&&(b>=c) then case2 else case3 where case1 = (a + (if b>=c then c else b)) / 2.0 case2 = (b + (if a>=c then c else a)) / 2.0 case3 = (c + (if a>=b then b else a)) / 2.0
Solução 2b
averageExt a b c = (if (a>=b)&&(a>=c) then case1 else if(b>=a)&&(b>=c) then case2 else case3) / 2.0 where case1 = (a + (if b>=c then c else b)) case2 = (b + (if a>=c then c else a)) case3 = (c + (if a>=b then b else a))

4. Formalization of Conceptual Plans (Knowledge Modeling)

Knowledge representation was developed upon first order logics because it is a language the allow us to express precisely the rationale for this investigation.

4.1. Organization

Each metaphorical variant of a problem has an associated list of concepts that are expected to be acquired (abstracted) by the students. Scripts are used to describe a solution and are modeled by syntactic components. Each solution has an identifier. We choose modeling the description of a solution through two relation symbols: *CognitiveComp* and *SyntacticComp*, both binary, representing association between a solution and a cognitive or syntactic component. Each syntactic component and each cognitive component will be described by a relation associating an identifier to a description. The first one is called *SyntacticElem* and the second *CognitiveElem*. The several cognitive or syntactic variations are modeled by the predicate symbols *TypeSyntacticVar* and *TypeCognitiveVar*, as follows:

CognitiveComp(<solution>,<cognitive component>)

SyntacticComp (<solution>,<syntactic component>)

SyntacticElem (<identifier>,<description>)

CognitiveElem (<identifier>,<description>)

TypeSyntacticVar (<sort of syntactic variation>)

TypeCognitiveVar (<sort of cognitive variation>)

Solution (<solution>)

Solutions are linked among themselves by one o more syntactic variations or by one or more cognitive variations. The first case is modeled by using the predicate symbol SyntacticVar and the second one by using CognitiveVar. In any relation, the two solutions, the component and the sort of variation are involved.

SyntacticVar(<solution1>, <solution2>, <syntactic component>, <sort of variation>)

CognitiveVar(<solution1>, <solution2>, <cognitive component>, <sort of variation>)

4.2. Axioms

We have identified some elements for a more detailed description of the knowledge present in the solutions described and their interactions. The following list is the result of an initial exploitation on these relationships and their peculiarities. We understand there is still much to be identified, and intend to do so in the near future.

1) syntactic variation is a non-reflexive relation

2) cognitive variation is a non-reflexive relation

3) syntactic variation is an anti-symmetrical relation

4) cognitive variation is an anti-symmetrical relation

5) Each solution cannot have more than one source of syntactic derivation

6) Each solution cannot have more than one source of cognitive derivation

7) Between two solutions, cognitive variations in a given component are unique

8) Between two solutions, syntactic variations in a given component, are unique

9) Every solution, except the abstract solution (solution 0), has a solution from which it is derived syntactly or semanticly

10) Abstract solutions have only metaphorical cognitive variants

11) Abstract solutions do not have syntactic variants.

4.2.1. Logical representation of Axioms

ax1: $\forall$s1, s2, c1, t1 SyntacticVar(s1,s2,c1,t1) $\Rightarrow$ $\neq$(s1,s2)

ax2: $\forall$s1, s2, c1, t1 CognitiveVar(s1,s2,c1,t1) $\Rightarrow$ $\neq$(s1,s2)

ax3: $\forall$s1, s2, c1, t1 SyntacticVar(s1,s2,c1,t1) $\Rightarrow$ ~$\exists$c2,t2 SyntacticVar(s2,s1,c2,t2)

ax4: $\forall$s1, s2, c1, t1 CognitiveVar(s1,s2,c1,t1) $\Rightarrow$ ~$\exists$c2,t2 CognitiveVar(s2,s1,c2,t2)

ax5: $\forall$s1, s2, s3, c1, c2, t1, t2 SyntacticVar(s1,s2,c1,t1) $\wedge$ SyntacticVar(s3,s2,c2,t2) $\Rightarrow$ =(s1,s3)

ax6: $\forall$s1, s2, s3, c1, c2, t1, t2 CognitiveVar(s1,s2,c1,t1) $\wedge$ CognitiveVar(s3,s2,c2,t2) $\Rightarrow$ =(s1,s3)

ax7: $\forall$s1, s2, s3, c1, c2, t1, t2 SyntacticVar(s1,s2,c1,t1) $\wedge$ SyntacticVar(s3,s2,c2,t2) $\Rightarrow$ =(c1,c2) $\wedge$ =(t1,t2)

ax8: $\forall$s1, s2, s3, c1, c2, t1, t2 CognitiveVar(s1,s2,c1,t1) $\wedge$ CognitiveVar(s3,s2,c2,t2) $\Rightarrow$ =(c1,c2) $\wedge$ =(t1,t2)

ax9: $\forall$s solution(s) $\wedge \neq$(s,solution-0) $\Rightarrow \exists$ s1,c1,t1 SyntacticVar(s1,s,c1,t1)

$\vee \exists$s2,c2,t2 CognitiveVar(s2,s,c2,t2)

ax10: $\forall$s,s1,c,t solution(s) $\wedge$ =(s,solution-0) $\wedge$ CognitiveVar(s,s1,c,t)

$\Rightarrow$ =(t,metaphor)

ax11: $\forall$s,s1,c,t solution(s) $\wedge$ =(s,solution-0) $\Rightarrow \neg \exists$s1, c, t SyntacticVar(s,s1,c,t)

4.2.2. Application of Axioms

The axioms presented here have been defined by analyzing performance of four groups of students during two academic semesters. We have used these initial results to other classes participating in this research and extended experimental procedures with logics tests, questionnaires and interviews.

The results obtained so far have confirmed the axioms described here and their validity to others concepts present in introductory programming courses, like the use of lists and recursion.

5. Concluding Remarks

In this work we have identified important elements from solutions presented by students of introductory programming courses within a functional approach. By comparing those solutions with a set considered more suitable from predetermined points of view, identification and classification of cognitive variants were made possible and have allowed us to model relationships between solutions using first order logic.

We believe this work may contribute for a methodology of programming education with emphasis on high readability a reuse of solutions. In this context, we could help students to compare their solutions in order to improve those aspects.

In addition to expansion on the set of axioms presented in Section 4.2 and preparation of supporting material for programming courses based on the ideas reported here, the development of tools based on the knowledge modeled as illustrated here, is also a natural consequence of this work.

References

Bird, R.S., Wadler, Ph., Introduction to Functional Programming, Prentice Hall, ISBN 0-13-484197-2, 1988, New York.

Bloom, B. S. (ed.) *Taxonomy of Educational Objectives: the classification of educational goals. Handbook I – cognitive domain*. 1956, New York.

Castro, T., Castro Jr, A. N., Menezes, C. S., Boeres, M. C. S., Rauber, M. C. P. V. Utilizando Programação Funcional em Disciplinas Introdutórias de Computação In: XXII Congresso da Sociedade Brasileira de Computação / X Workshop sobre Educação em Computação, 2002, Florianópolis.

Giegerich, R., Hinze R., Kurtz, S. Straight to the Heart of Computer Science via Functional Programming. Workshop on Functional and Declarative Programming in Education. 1999. Paris, FR.

Joosten, S., Van-den-Berg, K., Van-der-Hoeven, G. Teaching Functional Programming to First-Year Students. Journal of Functional Programming. Vol.3, N.1. 1993.

Mckeown, J., Farrell, T. Why We Need to Develop Success in Introductory Programming Courses. CCSC - Central Plains Conference. 1999. Maryville, MO.